

Chapter 6: More Conditionals and Loops

Lab Exercises

<u>Topics</u>	<u>Lab Exercises</u>
The switch statement	A Charge Account Statement Activities at Lake LazyDays Rock, Paper, Scissors Date Validation
Conditional Operator	Processing Grades
The do statement	More Guessing Election Day
The for statement	Finding Maximum and Minimum Values Counting Characters Using the Coin Class
Drawing with loops and conditionals	A Rainbow Program
Dialog Boxes	Modifying EvenOdd.java A Pay Check Program

A Charge Account Statement

Write a program to prepare the monthly charge account statement for a customer of CS CARD International, a credit card company. The program should take as input the previous balance on the account and the total amount of additional charges during the month. The program should then compute the interest for the month, the total new balance (the previous balance plus additional charges plus interest), and the minimum payment due. Assume the interest is 0 if the previous balance was 0 but if the previous balance was greater than 0 the interest is 2% of the total owed (previous balance plus additional charges). Assume the minimum payment is as follows:

new balance	for a new balance less than \$50
\$50.00	for a new balance between \$50 and \$300 (inclusive)
20% of the new balance	for a new balance over \$300

So if the new balance is \$38.00 then the person must pay the whole \$38.00; if the balance is \$128 then the person must pay \$50; if the balance is \$350 the minimum payment is \$70 (20% of 350). The program should print the charge account statement in the format below. Print the actual dollar amounts in each place using currency format from the `NumberFormat` class—see Listing 3.4 of the text for an example that uses this class.

```
CS CARD International Statement
=====

Previous Balance:      $
Additional Charges:    $
Interest:              $
New Balance:           $
Minimum Payment:       $
```

Activities at Lake LazyDays

As activity directory at Lake LazyDays Resort, it is your job to suggest appropriate activities to guests based on the weather:

```
temp >= 80:      swimming
60 <= temp < 80: tennis
40 <= temp < 60:  golf
temp < 40:       skiing
```

1. Write a program that prompts the user for a temperature, then prints out the activity appropriate for that temperature. Use a cascading if, and be sure that your conditions are no more complex than necessary.
2. Modify your program so that if the temperature is greater than 95 or less than 20, it prints “Visit our shops!”. (Hint: Use a boolean operator in your condition.) For other temperatures print the activity as before.

Rock, Paper, Scissors

Program *Rock.java* contains a skeleton for the game Rock, Paper, Scissors. Open it and save it to your directory. Add statements to the program as indicated by the comments so that the program asks the user to enter a play, generates a random play for the computer, compares them and announces the winner (and why). For example, one run of your program might look like this:

```
$ java Rock
Enter your play: R, P, or S

r
Computer play is S
Rock crushes scissors, you win!
```

Note that the user should be able to enter either upper or lower case r, p, and s. The user's play is stored as a string to make it easy to convert whatever is entered to upper case. Use a switch statement to convert the randomly generated integer for the computer's play to a string.

```
// *****
//   Rock.java
//
//   Play Rock, Paper, Scissors with the user
//
// *****
import java.util.Scanner;
import java.util.Random;

public class Rock
{
    public static void main(String[] args)
    {
        String personPlay;    //User's play -- "R", "P", or "S"
        String computerPlay;  //Computer's play -- "R", "P", or "S"
        int computerInt;      //Randomly generated number used to determine
                               //computer's play

        Scanner scan = new Scanner(System.in);
        Random generator = new Random();

        //Get player's play -- note that this is stored as a string
        //Make player's play uppercase for ease of comparison
        //Generate computer's play (0,1,2)
        //Translate computer's randomly generated play to string
        switch (computerInt)
        {

        }

        //Print computer's play
        //See who won. Use nested ifs instead of &&.
        if (personPlay.equals(computerPlay))
            System.out.println("It's a tie!");
        else if (personPlay.equals("R"))
            if (computerPlay.equals("S"))
                System.out.println("Rock crushes scissors. You win!!");
            else
                //... Fill in rest of code
        }
    }
}
```

Date Validation

In this exercise you will write a program that checks to see if a date entered by the user is a valid date in the second millenium. A skeleton of the program is in *Dates.java*. Open this program and save it to your directory. As indicated by the comments in the program, fill in the following:

1. An assignment statement that sets monthValid to true if the month entered is between 1 and 12, inclusive.
2. An assignment statement that sets yearValid to true if the year is between 1000 and 1999, inclusive.
3. An assignment statement that sets leap Year to true if the year is a leap year. Here is the leap year rule (there's more to it than you may have thought!):

If the year is divisible by 4, it's a leap year UNLESS it's divisible by 100, in which case it's not a leap year UNLESS it's divisible by 400, in which case it is a leap year. If the year is not divisible by 4, it's not a leap year.

Put another way, it's a leap year if a) it's divisible by 400, or b) it's divisible by 4 and it's *not* divisible by 100. So 1600 and 1512 are leap years, but 1700 and 1514 are not.

4. An if statement that determines the number of days in the month entered and stores that value in variable daysInMonth. If the month entered is not valid, daysInMonth should get 0. Note that to figure out the number of days in February you'll need to check if it's a leap year.
5. An assignment statement that sets dayValid to true if the day entered is legal for the given month and year.
6. If the month, day, and year entered are all valid, print "Date is valid" and indicate whether or not it is a leap year. If any of the items entered is not valid, just print "Date is not valid" without any comment on leap year.

```
// *****
// Dates.java
//
// Determine whether a 2nd-millennium date entered by the user
// is valid
// *****
import java.util.Scanner;

public class Dates
{
    public static void main(String[] args)
    {
        int month, day, year;    //date read in from user
        int daysInMonth;        //number of days in month read in
        boolean monthValid, yearValid, dayValid; //true if input from user is valid
        boolean leapYear;        //true if user's year is a leap year

        Scanner scan = new Scanner(System.in);

        //Get integer month, day, and year from user
        //Check to see if month is valid
        //Check to see if year is valid
        //Determine whether it's a leap year
        //Determine number of days in month
        //User number of days in month to check to see if day is valid
        //Determine whether date is valid and print appropriate message
    }
}
```

Processing Grades

The file *Grades.java* contains a program that reads in a sequence of student grades and computes the average grade, the number of students who pass (a grade of at least 60) and the number who fail. The program uses a loop (which you learn about in the next section).

1. Compile and run the program to see how it works.
2. Study the code and do the following.
 - ☐ Replace the statement that finds the sum of the grades with one that uses the `+=` operator.
 - ☐ Replace each of three statements that increment a counting variable with statements using the increment operator.
3. Run your program to make sure it works.
4. Now replace the “if” statement that updates the pass and fail counters with the conditional operator.

```
// *****
//  Grades.java
//
//  Read in a sequence of grades and compute the average
//  grade, the number of passing grades (at least 60)
//  and the number of failing grades.
//  *****
import java.util.Scanner;

public class Grades
{
    // -----
    //  Reads in and processes grades until a negative number is entered.
    //  -----
    public static void main (String[] args)
    {
        double grade; //a student's grade
        double sumOfGrades; // a running total of the student grades
        int numStudents; //a count of the students
        int numPass;      //a count of the number who pass
        int numFail;      // a count of the number who fail

        Scanner scan = new Scanner(System.in);

        System.out.println ("\nGrade Processing Program\n");

        // Initialize summing and counting variables
        sumOfGrades = 0;
        numStudents = 0;
        numPass = 0;
        numFail = 0;

        // Read in the first grade
        System.out.print ("Enter the first student's grade: ");
        grade = scan.nextDouble();

        while (grade >= 0)
        {
            sumOfGrades = sumOfGrades + grade;
            numStudents = numStudents + 1;

            if (grade < 60)
                numFail = numFail + 1;
            else
                numPass = numPass + 1;
        }
    }
}
```

```

        // Read the next grade
        System.out.print ("Enter the next grade (a negative to quit): ");
        grade = scan.nextDouble();
    }

    if (numStudents > 0)
    {
        System.out.println ("\nGrade Summary: ");
        System.out.println ("Class Average: " + sumOfGrades/numStudents);
        System.out.println ("Number of Passing Grades: " + numPass);
        System.out.println ("Number of Failing Grades: " + numFail);
    }
    else
        System.out.println ("No grades processed.");
}
}

```

More Guessing

File *Guess.java* contains the skeleton for a program that uses a while loop to play a guessing game. (This problem is described in the previous lab exercise.) Revise this program so that it uses a *do ... while* loop rather than a while loop. The general outline using a *do... while* loop is as follows:

```
// set up (initializations of the counting variables)
....

do
{
    // read in a guess
    ...

    // check the guess and print appropriate messages

    ...
}
while ( condition );
```

A key difference between a *while* and a *do... while* loop to note when making your changes is that the body of the *do ... while* loop is executed before the condition is ever tested. In the while loop version of the program, it was necessary to read in the user's first guess before the loop so there would be a value for comparison in the condition. In the *do... while* this "priming" read is no longer needed. The user's guess can be read in at the beginning of the body of the loop.

Election Day

It's almost election day and the election officials need a program to help tally election results. There are two candidates for office—Polly Tichen and Ernest Orator. The program's job is to take as input the number of votes each candidate received in each voting precinct and find the total number of votes for each. The program should print out the final tally for each candidate—both the total number of votes each received and the percent of votes each received. Clearly a loop is needed. Each iteration of the loop is responsible for reading in the votes from a single precinct and updating the tallies. A skeleton of the program is in the file *Election.java*. Open a copy of the program in your text editor and do the following.

1. Add the code to control the loop. You may use either a while loop or a do...while loop. The loop must be controlled by asking the user whether or not there are more precincts to report (that is, more precincts whose votes need to be added in). The user should answer with the character y or n though your program should also allow uppercase responses. The variable *response* (type String) has already been declared.
2. Add the code to read in the votes for each candidate and find the total votes. Note that variables have already been declared for you to use. Print out the totals and the percentages after the loop.
3. Test your program to make sure it is correctly tallying the votes and finding the percentages AND that the loop control is correct (it goes when it should and stops when it should).
4. The election officials want more information. They want to know how many precincts each candidate carried (won). Add code to compute and print this. You need three new variables: one to count the number of precincts won by Polly, one to count the number won by Ernest, and one to count the number of ties. Test your program after adding this code.

```
// *****
// Election.java
//
// This file contains a program that tallies the results of
// an election. It reads in the number of votes for each of
// two candidates in each of several precincts. It determines
// the total number of votes received by each candidate, the
// percent of votes received by each candidate, the number of
// precincts each candidate carries, and the
// maximum winning margin in a precinct.
// *****

import java.util.Scanner;

public class Election
{
    public static void main (String[] args)
    {
        int votesForPolly; // number of votes for Polly in each precinct
        int votesForErnest; // number of votes for Ernest in each precinct
        int totalPolly; // running total of votes for Polly
        int totalErnest; // running total of votes for Ernest
        String response; // answer (y or n) to the "more precincts" question

        Scanner scan = new Scanner(System.in);

        System.out.println ();
        System.out.println ("Election Day Vote Counting Program");
        System.out.println ();

        // Initializations

        // Loop to "process" the votes in each precinct

        // Print out the results
    }
}
```

Finding Maximum and Minimum Values

A common task that must be done in a loop is to find the maximum and minimum of a sequence of values. The file *Temps.java* contains a program that reads in a sequence of hourly temperature readings over a 24-hour period. You will be adding code to this program to find the maximum and minimum temperatures. Do the following:

1. Save the file to your directory, open it and see what's there. Note that a *for* loop is used since we need a count-controlled loop. Your first task is to add code to find the maximum temperature read in. In general to find the maximum of a sequence of values processed in a loop you need to do two things:
 - You need a variable that will keep track of the maximum of the values processed so far. This variable must be initialized before the loop. There are two standard techniques for initialization: one is to initialize the variable to some value *smaller* than any possible value being processed; another is to initialize the variable to the first value processed. In either case, after the first value is processed the maximum variable should contain the first value. For the temperature program declare a variable *maxTemp* to hold the maximum temperature. Initialize it to -1000 (a value less than any legitimate temperature).
 - The maximum variable must be updated each time through the loop. This is done by comparing the maximum to the current value being processed. If the current value is larger, then the current value is the new maximum. So, in the temperature program, add an *if* statement inside the loop to compare the current temperature read in to *maxTemp*. If the current temperature is larger set *maxTemp* to that temperature. NOTE: If the current temperature is NOT larger, DO NOTHING!
2. Add code to print out the maximum after the loop. Test your program to make sure it is correct. Be sure to test it on at least three scenarios: the first number read in is the maximum, the last number read in is the maximum, and the maximum occurs somewhere in the middle of the list. For testing purposes you may want to change the *HOURS_PER_DAY* variable to something smaller than 24 so you don't have to type in so many numbers!
3. Often we want to keep track of more than just the maximum. For example, if we are finding the maximum of a sequence of test grades we might want to know the name of the student with the maximum grade. Suppose for the temperatures we want to keep track of the time (hour) the maximum temperature occurred. To do this we need to save the current value of the *hour* variable when we update the *maxTemp* variable. This of course requires a new variable to store the time (hour) that the maximum occurs. Declare *timeOfMax* (type *int*) to keep track of the time (hour) the maximum temperature occurred. Modify your *if* statement so that in addition to updating *maxTemp* you also save the value of *hour* in the *timeOfMax* variable. (WARNING: you are now doing TWO things when the *if* condition is TRUE.)
4. Add code to print out the time the maximum temperature occurred along with the maximum.
5. Finally, add code to find the minimum temperature and the time that temperature occurs. The idea is the same as for the maximum. NOTE: Use a separate *if* when updating the minimum temperature variable (that is, don't add an *else* clause to the *if* that is already there).

```

// *****
// Temps.java
//
// This program reads in a sequence of hourly temperature
// readings (beginning with midnight) and prints the maximum
// temperature (along with the hour, on a 24-hour clock, it
// occurred) and the minimum temperature (along with the hour
// it occurred).
// *****

import java.util.Scanner;

public class Temps
{
    // -----
    // Reads in a sequence of temperatures and finds the
    // maximum and minimum read in.
    // -----
    public static void main (String[] args)
    {
        final int HOURS_PER_DAY = 24;

        int temp;    // a temperature reading

        Scanner scan = new Scanner(System.in);

        // print program heading
        System.out.println ();
        System.out.println ("Temperature Readings for 24 Hour Period");
        System.out.println ();

        for (int hour = 0; hour < HOURS_PER_DAY; hour++)
        {
            System.out.print ("Enter the temperature reading at " + hour +
                             " hours: ");
            temp = scan.nextInt();
        }

        // Print the results
    }
}

```

Counting Characters

The file *Count.java* contains the skeleton of a program to read in a string (a sentence or phrase) and count the number of blank spaces in the string. The program currently has the declarations and initializations and prints the results. All it needs is a loop to go through the string character by character and count (update the *countBlank* variable) the characters that are the blank space. Since we know how many characters there are (the *length* of the string) we use a count controlled loop—*for* loops are especially well-suited for this.

1. Add the *for* loop to the program. Inside the *for* loop you need to access each individual character—the *charAt* method of the *String* class lets you do that. The assignment statement

```
ch = phrase.charAt(i);
```

assigns the variable *ch* (type *char*) the character that is in index *i* of the *String phrase*. In your *for* loop you can use an assignment similar to this (replace *i* with your loop control variable if you use something other than *i*). NOTE: You could also directly use *phrase.charAt(i)* in your *if* (without assigning it to a variable).

2. Test your program on several phrases to make sure it is correct.
3. Now modify the program so that it will count several different characters, not just blank spaces. To keep things relatively simple we'll count the a's, e's, s's, and t's (both upper and lower case) in the string. You need to declare and initialize four additional counting variables (e.g. *countA* and so on). Your current *if* could be modified to cascade but another solution is to use a *switch* statement. Replace the current *if* with a *switch* that accounts for the 9 cases we want to count (upper and lower case a, e, s, t, and blank spaces). The cases will be based on the value of the *ch* variable. The *switch* starts as follows—complete it.

```
switch (ch)
{
    case 'a':
    case 'A':    countA++;
                break;

    case ....
}
```

Note that this *switch* uses the “fall through” feature of *switch* statements. If *ch* is an ‘a’ the first case matches and the *switch* continues execution until it encounters the *break* hence the *countA* variable would be incremented.

4. Add statements to print out all of the counts.
5. It would be nice to have the program let the user keep entering phrases rather than having to restart it every time. To do this we need another loop surrounding the current code. That is, the current loop will be nested inside the new loop. Add an outer *while* loop that will continue to execute as long as the user does NOT enter the phrase *quit*. Modify the prompt to tell the user to enter a phrase or *quit* to quit. Note that all of the initializations for the counts should be inside the *while* loop (that is we want the counts to start over for each new phrase entered by the user). All you need to do is add the *while* statement (and think about placement of your reads so the loop works correctly). Be sure to go through the program and properly indent after adding code—with nested loops the inner loop should be indented.

```

// *****
// Count.java
//
// This program reads in strings (phrases) and counts the
// number of blank characters and certain other letters
// in the phrase.
// *****

import java.util.Scanner;

public class Count
{
    public static void main (String[] args)
    {
        String phrase;    // a string of characters
        int countBlank;    // the number of blanks (spaces) in the phrase
        int length;        // the length of the phrase
        char ch;           // an individual character in the string

        Scanner scan = new Scanner(System.in);

        // Print a program header
        System.out.println ();
        System.out.println ("Character Counter");
        System.out.println ();

        // Read in a string and find its length
        System.out.print ("Enter a sentence or phrase: ");
        phrase = scan.nextLine();
        length = phrase.length();

        // Initialize counts
        countBlank = 0;

        // a for loop to go through the string character by character
        // and count the blank spaces

        // Print the results
        System.out.println ();
        System.out.println ("Number of blank spaces: " + countBlank);
        System.out.println ();
    }
}

```

Using the Coin Class

The Coin class from Listing 4.2 in the text is in the file *Coin.java*. Copy it to your directory, then write a program to find the length of the longest run of heads in 100 flips of the coin. A skeleton of the program is in the file *Runs.java*. To use the Coin class you need to do the following in the program:

1. Create a coin object.
2. Inside the loop, you should use the *flip* method to flip the coin, the *toString* method (used implicitly) to print the results of the flip, and the *getFace* method to see if the result was HEADS. Keeping track of the current run length (the number of times in a row that the coin was HEADS) and the maximum run length is an exercise in loop techniques!
3. Print the result after the loop.

```
// *****
// Coin.java          Author: Lewis and Loftus
//
// Represents a coin with two sides that can be flipped.
// *****

public class Coin
{
    public final int HEADS = 0;
    public final int TAILS = 1;

    private int face;

    // -----
    // Sets up the coin by flipping it initially.
    // -----
    public Coin ()
    {
        flip();
    }

    // -----
    // Flips the coin by randomly choosing a face.
    // -----
    public void flip()
    {
        face = (int) (Math.random() * 2);
    }

    // -----
    // Returns true if the current face of the coin is heads.
    // -----
    public boolean isHeads()
    {
        return (face == HEADS);
    }

    // -----
    // Returns the current face of the coin as a string.
    // -----
    public String toString()
    {
        String faceName;
        if (face == HEADS)
            faceName = "Heads";
```

```

        else
            faceName = "Tails";
        return faceName;
    }
}

// *****
// Runs.java
//
// Finds the length of the longest run of heads in 100 flips of a coin.
// *****

public class Runs
{
    public static void main (String[] args)
    {
        final int FLIPS = 100; // number of coin flips

        int currentRun =0; // length of the current run of HEADS
        int maxRun =0;      // length of the maximum run so far

        // Create a coin object

        // Flip the coin FLIPS times
        for (int i = 0; i < FLIPS; i++)
        {
            // Flip the coin & print the result

            // Update the run information

        }
        // Print the results
    }
}

```

A Rainbow Program

Write a program that draws a rainbow. (This is one of the Programming Projects at the end of Chapter 6 in the text.) As suggested in the text, your rainbow will be concentric arcs, each a different color. The basic idea of the program is similar to the program that draws a bull's eye in Listing 6.5 and 6.6 of the text. You should study that program and understand it before starting your rainbow. The major difference in this program (other than drawing arcs rather than circles) is making the different arcs different colors. You can do this in several different ways. For example, you could have a variable for the color code, initialize it to some value then either increment or decrement by some amount each pass through the loop (you'll need to experiment with numbers to see the effect on the color). Or you could try using three integer variables—one for the amount of red, one for the amount of green, and the other for the amount of blue—and modify these (by adding or subtracting some amounts) each time through the loop (each of these must be an integer in the range 0 - 255). For this technique you would need to use the constructor for a Color object that takes three integers representing the amount of red, green, and blue as parameters. Other possibilities are to make each arc a different random color, or have set colors (use at least 4 different colors) and cycle through them (using an idea similar to the way the bull's eye program switches between black and white).

Modifying *EvenOdd.java*

File *EvenOdd.java* contains the dialog box example in Listing 6.9 the text.

1. Compile and run the program to see how it works.
2. Write a similar class named *SquareRoots* (you may modify *EvenOdd*) that computes and displays the square root of the integer entered.

```
//*****
// EvenOdd.java Author: Lewis/Loftus
//
// Demonstrates the use of the JOptionPane class.
//*****

import javax.swing.JOptionPane;

class EvenOdd
{
    //-----
    // Determines if the value input by the user is even or odd.
    // Uses multiple dialog boxes for user interaction.
    //-----
    public static void main (String[] args)
    {
        String numStr, result;
        int num, again;

        do
        {
            numStr = JOptionPane.showInputDialog ("Enter an integer: ");

            num = Integer.parseInt(numStr);

            result = "That number is " + ((num%2 == 0) ? "even" : "odd");

            JOptionPane.showMessageDialog (null, result);

            again = JOptionPane.showConfirmDialog (null, "Do Another?");
        }
        while (again == JOptionPane.YES_OPTION);
    }
}
```

A Pay Check Program

Write a class *PayCheck* that uses dialog boxes to compute the total gross pay of an hourly wage worker. The program should use input dialog boxes to get the number of hours worked and the hourly pay rate from the user. The program should use a message dialog to display the total gross pay. The pay calculation should assume the worker earns time and a half for overtime (for hours over 40).